Received: 1 August 2018 | Revised: 16 May 2019 | Accepted: 22 May 2019
DOI: 10.1002/cpe.5443

WILEY

SPECIAL ISSUE PAPER

An unstructured CFD mini-application for the performance prediction of a production CFD code

A. M. B. Owenson[1] | S. A. Wright[2] | R. A. Bunt[1] | Y. K. Ho[3] | M. J. Street[3] | S. A. Jarvis[1]

[1]Department of Computer Science, University of Warwick, Coventry, UK
[2]Department of Computer Science, University of York, York, UK
[3]Design Systems Engineering, Rolls-Royce plc, Derby, UK

Correspondence
A. M. B. Owenson, Department of Computer Science, University of Warwick, Coventry CV4 7AL, UK.
Email: a.m.b.owenson@warwick.ac.uk

Funding information
Rolls-Royce plc; EU Horizon 2020 Clean Sky Project; UK Engineering and Physical Sciences Research Council (EPSRC), Grant/Award Number: EP/S005072/1; Intel Corporation, Grant/Award Number: 15220082

Summary

Maintaining the performance of large scientific codes is a difficult task. To aid in this task, a number of mini-applications have been developed that are more tractable to analyze than large-scale production codes while retaining the performance characteristics of them. These "mini-apps" also enable faster hardware evaluation and, for sensitive commercial codes, allow evaluation of code and system changes outside of access approval processes. In this paper, we develop MG-CFD, a mini-application that represents a geometric multigrid, unstructured computational fluid dynamics (CFD) code, designed to exhibit similar performance characteristics without sharing commercially sensitive code. We detail our experiences of developing this application using guidelines detailed in existing research and contributing further to these. Our application is validated against the inviscid flux routine of HYDRA, a CFD code developed by Rolls-Royce plc for turbomachinery design. This paper (1) documents the development of MG-CFD, (2) introduces an associated performance model with which it is possible to assess the performance of HYDRA on new HPC architectures, and (3) demonstrates that it is possible to use MG-CFD and the performance models to predict the performance of HYDRA with a mean error of 9.2% for strong-scaling studies.

KEYWORDS

computational fluid dynamics, high performance computing, mini-application, performance analysis, performance modeling, scientific computing

1 | INTRODUCTION

The rapid development of new hardware and software in High Performance Computing (HPC) is greatly benefiting scientific discovery; with each new development comes new opportunities for improving the performance of scientific applications. Evaluating the potential improvements offered by these developments is often a time consuming process due to the complexity of the applications involved, and the learning curve for new machines, architectures, and toolchains.

In recognition of these challenges, many HPC centers are turning in supporting tools and methodologies (eg, predictive performance modeling[1-4] and hardware simulation[5,6]) to evaluate new systems ahead of procurement. Additionally, mini-applications have been shown to facilitate rapid evaluation of new hardware and programming techniques; these applications capture the key performance characteristics of a parent code in a much more concise form; making them easier to work with than full production codes but equally useful in performance engineering activities. The use of mini-applications has been well documented[7-10] and has spawned several suites of such applications[11,12] for industry and research community to examine. Recent use of mini-apps includes the recently established ASiMoV strategic partnership between Rolls-Royce plc and five leading UK universities, whose aim is to achieve the first high-fidelity simulation of a complete and operating gas-turbine engine.[13] This will require several breakthroughs including achieving exascale performance of a CFD simulation code.

This paper extends previous work on the development and early validation of a geometric multigrid, unstructured grid Computational Fluid Dynamics (CFD) mini-application.[14] In this paper, we refine and conclude this development and show, through new research, how it can be used for performance prediction. In so doing, we address a limitation of our previous work, where the mini-application had a greater arithmetic intensity than the target kernel. This presented two significant issues that this paper seeks to address: (1) computational speed-ups identified with the mini-app such as vectorization may not transfer to the target code, and (2) the mini-app may not identify optimizations of, or improvements to, memory bandwidth that would benefit the target code.

Specifically, this paper makes the following contributions:

- This paper reports the development, refinement, and testing of MG-CFD, the only multigrid unstructured finite-volume CFD mini-application.
- The MG-CFD has been developed as part of a long-standing university/industry collaboration and, as a result, is the representative of the production code HYDRA, which is the primary CFD code used by Rolls-Royce plc for turbomachinery design.
- This paper presents a new performance projection model for HYDRA, with which it is possible to project from MG-CFD to HYDRA performance on a range of existing and emerging HPC architectures. This is highly significant for Rolls-Royce plc as they increase their use of virtual certification and simulation-based engine design.
- This paper demonstrates that it is possible to use a mini-application and performance modeling to predict the performance of a production "target" code, with a mean error of 9.2% for strong-scaling studies.

This paper is structured as follows. In Section 2, we discuss related work. In Section 3, we summarize the functionality of HYDRA, which we aim to capture within the mini-application. In Section 4, we describe our experiences constructing MG-CFD. In Section 5, we validate the performance characteristics of MG-CFD when compared to the target kernel. In Section 6, we describe the proposed analytical model, and validate it on several HPC systems. Finally, Section 7 concludes this paper.

2 | RELATED WORK

There are numerous benchmarks and mini-applications representing the performance of different classes of HPC applications, some of which have been released as component parts of projects such as the Mantevo Project,[11] the ECP Proxy Apps Suite,[15] and the UK Mini-App Consortium.[12] Mini-applications from these repositories have been used in a variety of contexts.

One such example is miniMD, which has been used to explore the performance of molecular dynamics codes on the Intel Xeon Phi Knights Corner architecture.[7] Using a combination of AVX intrinsics and algorithmic optimizations, eg, overlapping PCIe transfers with computation, the authors demonstrate a 5× speed-up for the gather-scatter bottleneck typically present in MD codes.

Mallinson et al compare the performance of two PGAS programming models (OpenSHMEM and Co-Array Fortran) against MPI using CloverLeaf, a Lagrangian-Eulerian hydrodynamics mini-application.[10] The authors demonstrate that OpenSHMEM is able to outperform an equivalent MPI implementation by 7.78 iterations/seconds, at 4096 sockets, when using proprietary nonblocking operations from Cray and 4 MB memory pages.

LULESH, a hydrodynamics mini-application representative of ALE3D, is used to assess the suitability of emerging parallel programming models (eg, Liszt and Loci) along with more established models such as OpenMP,[16] in terms of programmer productivity, runtime performance, and ease of optimization. The reduced size of LULESH when compared with ALE3D allowed the authors to examine eight parallel programming models. Their conclusion highlights that while the emerging models such as Chapel and Loci enable a high level of productivity, they cannot match the performance of more established models such as MPI and OpenMP.

Similarly, Reguly et al examine the performance of OP2, a domain specific framework for unstructured grid codes using the AIRFOIL CFD mini-application.[9] The authors demonstrate that they are able to achieve performance within 6% of a hand-coded implementation.

The CFD code included in the Rodinia benchmark suite has been used to examine the performance of a Graphics Processing Unit (GPU) when running unstructured grid applications.[17] From the results, Corrigan et al conclude that GPUs show promise for this class of code given an increase in double precision performance in the future.

The research in this paper similarly develops and makes use of a mini-application; however, our application additionally contains a geometric multigrid solver and supports mesh structures with variable node degree. The HPGMG-FV and LULESH mini-applications are most similar to our mini-application; however, the former operates on a structured mesh[18] and the latter does not have a multigrid solver.

Another body of work that is similar to our own and that we build upon deals with the validation of a mini-application's performance in relation to that of the parent code. The technique employed by Tramm et al involves comparing the correlation of parallel efficiency loss to performance counters for both the mini-application and the target code.[8] Previously, this technique has been applied to mini-applications of a neutron transport code[8]; we employ and validate this technique on a different class of application. Messer et al develop three mini-applications and use a comparison between the scalability of the mini-application and the original code as evidence of their similarity.[19] However, the authors focus on distributed memory scalability, in this research, we focus on intra-node shared memory scalability.

Finally, in this paper, we explore how to project from mini-application performance to predict that of the target code. Sharkawi et al propose a technique of identifying surrogate codes that are quantifiably similar to the target code according to 25 performance metrics.[20] These surrogates are executed, a genetic algorithm selects a weighting, and their weighted average is used as a performance prediction, achieving a mean error of

7.2% on a IBM Power6 and 10.5% on a Intel Core. Hoste et al apply a similar technique but to microarchitecture-independent metrics, predicting the ranking of machine performance with 0.89 mean rank correlation.[21] In contrast, we show that an analytical model of the performance difference between a mini-application and its target code can provide projections of similar accuracy, with a mean error of 9.2% for strong-scaling studies.

3 | BACKGROUND

3.1 | HYDRA

The manufacturing industry is increasingly making use of CFD simulation codes to aid in the design and testing process of new products. One such code is HYDRA,[22] a suite of nonlinear, linear, and adjoint solvers developed by Rolls-Royce plc in collaboration with a number of UK universities. These solvers target airflow within turbomachinery, where the flow must be modeled as compressible, viscous, and turbulent. As such, they solve the Reynolds-Averaged form of the compressible Navier-Stokes equations, which embody conservations of mass, momentum, and energy. Turbulence modeling is enhanced with the Spalart-Allmaras one-equation model.[23] Equations are discretized using a MUSCL-based flux-differencing scheme, then block Jacobi preconditioned.[24] An explicit five-stage Runge-Kutta scheme is applied to improve stability in high viscosity regions and convergence of the multigrid method.[25] For more information on the background and numerical implementation of HYDRA, we refer the reader to the work of Lapworth.[22]

In this paper, we focus on HYDRA's nonlinear solver, which is summarized in Listing 1. A breakdown of HYDRA runtime by loop is given in Table 1, which shows that the two flux routines (`vflux` and `iflux`) account for almost half of the runtime on a single Xeon Broadwell node. Thus, we direct the development of the mini-application toward the goal of understanding and improving the performance of these routines. The two routines are computationally very similar, performing an integration of cell volume surface fluxes, but the `iflux` kernel is much smaller and so easier to detach from HYDRA. Mini-application development time can therefore be reduced by initially targeting the `iflux` routine.

3.2 | Multigrid

The HYDRA employs multigrid methods, which are designed to increase the rate of convergence for iterative solvers, and possess a useful computational property; the amount of computational work required is linear in the number of unknowns.[26] Multigrid applications operate on a

```
call jacob // Jacobi precondition
for iter = 1 to niter do
  for level in [0, 1, 2, 3, 2, 1] do
    for timestep = 1 to 5 do
      if dissipative flux update then
        call grad // Gradient
        call vflux // Viscous flux
        if viscous wall then
          call wfflux // Viscous wall flux
        end if
        call wvflux // Viscous near-wall flux
      end if

      call iflux // Invisid flux
      call srcsa // Spalart-Allmaras source term
      call update // Update flow solution
    end for

    /* transfer solution up/down multigrid */
    if direction = up then
      call restrict
    else
      call prolong
    end if
  end for
end for
```

Listing 1 HYDRA solver pseudo-code, with V-cycle geometric multigrid

Loop	Function	Runtime %
JACOB	Jacobi preconditioner matrices	6.8
GRAD	Gradient	16.8
VFLUX	Viscous fluxes	35.8
IFLUX	Inviscid fluxes	10.7
SRCSA	Spalart-Allmaras source term	14.6
UPDATE	Update flow	7.3
—	Other routines	8.0

TABLE 1 HYDRA runtime breakdown on single node Xeon Broadwell with 28 MPI processes

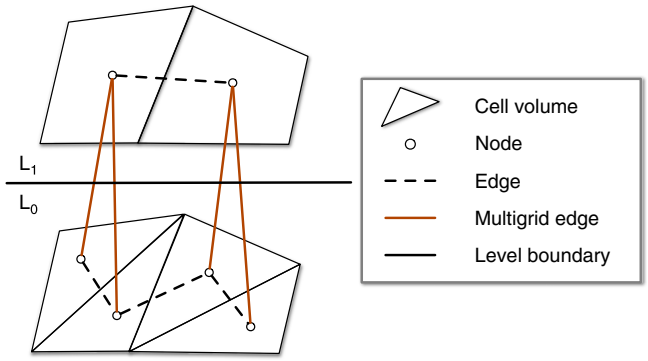

FIGURE 1 Representation of a finite-volume decomposition mapped to an unstructured grid over two multigrid levels

hierarchy of grid levels; in this paper, we are concerned with geometric multigrid, wherein each grid level has its own explicit mesh geometry, and the coarse levels of the hierarchy are derived from the geometry of the finest level.

Starting at the finest level, multigrid applications use an iterative *smoothing* subroutine to reduce high frequency errors. Low frequency errors are then transferred to the next coarsest level (*restriction*), where they appear as high frequency errors and can thus be more rapidly smoothed by the same subroutine. Error corrections from the smoothing of coarse levels are then transferred back to finer levels (*prolongation*). The order in which prolongations and restrictions are applied is know as a cycle, of which this paper considers a single type; the so-called V-cycle.

There are several performance implications of using a geometric multigrid solver. First, there is the increased memory requirement of explicitly representing the geometries of all levels of the multigrid. Second, there are the additional irregular memory accesses from prolonging and restricting corrections between levels of the multigrid. Third, the coarsened meshes have reduced spatial locality.

3.3 | Unstructured grid

The HYDRA represents its aerospace models using an unstructured grid; with reference to Figure 1, an unstructured grid is a collection of nodes and edges, with the nodes being at an arbitrary position in space. For a cell-centered finite-volume decomposition, each node represents a cell and each edge represents a surface between two adjacent cells. Since HYDRA operates on multigrid datasets, there are also edges between related nodes of adjacent grid levels. The flexibility of the unstructured grid allows complex geometries to be represented and regions of interest to be denoted by increasing the density of nodes (cells) in these areas.

The neighbors of a node in an unstructured grid are not implicitly defined, as is the case for a structured mesh code where the neighbors can be determined using offsets to the array indices. This means that an explicit list of neighbors must be maintained so that when computation over nodes is performed (eg, the accumulation of fluxes), data can be read from the required locations. This of course has implications for the memory access pattern as there is no guarantee that a node's neighbors directly and regularly succeed it in memory.

4 | MINI-APPLICATION DEVELOPMENT

Although the benefits of mini-applications are increasingly documented (see Section 2), their development is not a well-defined process as it depends largely on their intended purpose.[19] This makes their development challenging as the purpose may differ on a project-by-project basis, limiting the reuse of technique and effort. However, considerations and guidelines are aggregated by Messer et al and summarized here as a set of questions for reference.[19]

1. Where does the application spend most of its execution time?
2. What performance characteristics will the mini-application capture?
3. Can any part of the development process be automated?
4. How can the build system be made as simple as possible?

The aim of these questions is to focus attention on (1) which aspects of the target code the mini-application should include, and (2) the components of the supporting configuration (eg, tools and datasets). We apply these guidelines to the development of MG-CFD and because the development of each mini-application is essentially unique, we consider it a valuable exercise to document the findings of this approach. Additionally, we add our own considerations to this list, which come from our own experiences of developing this and other mini-apps.

We address the first question in Section 3, the most time consuming regions of code are contained within the two flux routines (`vflux` and `iflux`), and it is these routines that we therefore focus on capturing within our mini-application. These kernels have the same computational structure, a single loop over edges, accumulating fluxes. In this work, we focus on `iflux` as it captures the memory access behavior of the unstructured grid while containing less code than `vflux`.

The second question is addressed by considering the purpose of our mini-app; to evaluate the potential impact of code optimizations, new parallel programming models and new hardware features on the computational performance of HYDRA. This use case suggests constructing

a mini-application that ignores I/O and inter-node communication costs and focuses only on computation, encouraging us to focus on more specific regions of the code.

Next, we propose our own consideration: which aspects of the target (eg, unstructured grid, finite volume, multigrid) contribute to the compute behavior within the most expensive regions of the code? This decomposition by simulation feature provides us with a route for including performance characteristics within the mini-application. Drawing upon other's experiences with HYDRA along with our own, we know that it is the irregular memory accesses that contribute greatly to the difficulty of running on different compute architectures. These irregular memory accesses come from two main sources: the edge updates over the unstructured grid and the restriction and prolongation of corrections between the multigrid levels (see Section 3.2).

4.1 | Implementation

With these features in mind, we base our mini-app on an existing code as (1) it is open source, so it will not be restricted in terms of where it can be run and (2) it shares simulation features with HYDRA.[17] We base our code on the CFD application by Corrigan et al, now included in the Rodinia benchmark suite.[27] We extend this code with the addition of multigrid, hence we name our mini-app MG-CFD. The existing code, written in C++, implements a three-dimensional finite-volume discretization of the Euler equations for inviscid, compressible flow over an unstructured grid. It performs a sweep over edges to accumulate fluxes, implemented as a loop over cell volumes with an inner loop over the edges between each cell and its neighbors. The `iflux` also performs a sweep over edges, but implements this as a single loop over all edges in the grid. Although these two loop schemes implement the same numerical method, the different loop structures can lead to different performance characteristics, particularly regarding parallelization. To increase similarity to `iflux`, we replace the existing nested loop with a single loop over all edges.

The resulting kernel differs from `iflux` only in the exact arithmetic operations performed; it is not possible for MG-CFD to perform the same arithmetic as `iflux` as this would mean subjecting MG-CFD to the same commercial portability restrictions as HYDRA itself. We further extend our mini-app with additional simulation features present in HYDRA. It should be noted that we do not focus on verifying the correctness of the simulation against a standard problem in this paper, as we are primarily interested in performance characteristics that we validate in Section 5.

Support for the computational behaviors of multigrid was implemented by augmenting the construction of the Euler solver presented by Corrigan et al with crude operators to transfer the state of the simulation between the levels of the multigrid. These operators are defined by Equations (1) and (2), which serve as restriction (fine to coarse grid) and prolongation (coarse to fine grid) operators, respectively,[28] where u_j^l represents simulation property u of node j at level l, and N_j^l is the set of node indices that are linked to node j at level l from $l-1$ of the grid.

$$u_j^l = \frac{\sum_{i \in N_j^l} u_i^{l-1}}{|N_j^l|} \tag{1}$$

$$u_{i \in N_j^l}^{l-1} = u_j^l. \tag{2}$$

The restriction operator (Equation (1)) primes the simulation properties with an average across nodes from the finer grid level, this mapping between levels is defined as part of the input deck. The prolongation operator (Equation (2)) reverses restriction by injecting the values from the coarse grid to the fine grid as dictated by the mapping.

The final code change is made to allow an arbitrary number of neighbors rather than the fixed four in the flux summation. The summation is already weighted by the surface area of the interface between nodes in the mesh, so no correction to the underlying mathematics is necessary to support this change.

4.1.1 | Supporting tools

Part of what makes a mini-application a useful tool is its simplicity. This, however, is not only restricted to the application itself and must also apply to the processes surrounding the mini-application and target application (eg, building, job submission).

We opt to simplify the building process by removing all reliance on third-party libraries such as the Hierarchical Data Format 5 (HDF5) library and the communications library. These can both be safely removed as the purpose of this mini-application is not to investigate I/O performance, inter-node communication performance, nor the overheads introduced by library abstractions. Removing these dependencies allows the application to be built swiftly with minor adjustment of compiler and its flags in the Makefile. Another challenge to consistent benchmarking is the need to create job submissions scripts, so we include examples of these for several common schedulers: SLURM, LSF, and Moab.

Utilities have been included to validate the final state of the simulation after changes to the configuration (eg, compiler flags, code optimizations, porting to accelerators) of the code. Additionally, we include tools to extract the geometries from the datasets used to prime HYDRA and transform them into a form, which is understood by the mini-application. We do this to reduce the number of factors, which could cause differences in runtime behavior between HYDRA and its mini-application.

5 | VALIDATION

In this section, we present a validation of our mini-application against the target code's behavior on a dual-socket 28-core Xeon Skylake node. Full hardware details are provided in Table 2.

The unstructured grid used for validation is derived from the geometry of Whittle Laboratory's low pressure axial flow turbine rotor cascade, a mesh of 105 K nodes, and 305 K edges representing a single rotor root section (blade and hub connection).[29] To aid visualization, a rotor section of NASA's SSME 2-stage fuel turbine is shown in Figure 2, consisting of multiple root sections with similar structure to the mesh we use.[30] The mesh is duplicated in memory by a factor of 120, producing a set of 120 disconnected meshes. Each kernel will then process each of the 120 meshes in turn. This ensures that the workload does not fit in the cache and enables multi-threaded execution at particular process counts such that no two threads work on the same mesh. The nodes have been renumbered by the Cuthill-McKee (CM) algorithm, and then the edges reordered by the new values of their endpoints. This ensures consistency with other performance analyses of unstructured mesh compute.

In this paper, MG-CFD is validated using two existing methods. First, we compare the OpenMP parallel efficiency of both `iflux` and MG-CFD for each level of the multigrid (MG).[19] Figure 3 presents the scaling performance of both codes on the finest MG mesh, showing that both codes suffer similar parallel efficiency loss up to 12 threads, after which `iflux` suffers loss at a greater rate than MG-CFD (scaling of the other MG levels is similar so these are not shown). This is a result of `iflux` having a significantly lower arithmetic intensity than MG-CFD, becoming memory-bound at a lower thread count. This emphasizes the importance of considering and accounting for differences between a mini-application and its target code when interpreting performance data. Similarity between scaling behavior does not imply that the underlying causes of the observed behavior are the same, and so we strengthen the comparison using a second approach. This involves comparing the correlation of parallel efficiency loss to performance counters for both the mini-application and the target code.[8]

It should be noted that we compare MG-CFD against a direct Fortran-to-C port of `iflux`, rather than the original Fortran implementation. We do this to ease and remove the effects of language from the comparison process; arguably this moves us further away from the true performance characteristics of the target code, but it still allows the examination of language independent features, such as memory access patterns and arithmetic intensity.

The PAPI library is used to collect performance counter data, which provides easy access to available performance counters and additionally defines a set of 108 "preset" counters that include performance counters typically found in many processors.[31] Figure 4 shows the correlation between each PAPI preset performance counter and parallel inefficiency. To account for variance of performance counters between runs, the mean of three measurements is used. For most of these events, the difference in correlation between MG-CFD and `iflux` is less than 0.1, indicating that both codes share many performance characteristics, but there are several differences in correlations that we address here. The correlations for the events PAPI_SR_INS, PAPI_LST_INS and PAPI_LD_INS differ by 0.2, with the correlation being stronger for `iflux`. These events count store and load micro-ops, so `iflux` being more sensitive to these is in agreement with it having the lower arithmetic intensity. A similar difference between correlations can be seen in the branching related events (PAPI_BR_$*$), but neither code performs branching operations within the loop body so these are considered to be false positive. The only large difference is with events relating to L1 cache misses (PAPI_L1_DCM, PAPI_L1_TCM, PAPI_L2_DCW, and PAPI_L2_TCW), for which a strong correlation is only present with MG-CFD. This is likely a consequence of the register spilling that occurs with MG-CFD but not `iflux`, effectively reserving some of the L1 cache for register values, which leaves less for reuse of mesh data.

TABLE 2 Hardware/software configurations

	Hardware		
	Broadwell	**Skylake**	**Knights Landing**
Model	Intel Xeon E5-2660 v4	Intel Xeon Silver 4116	Intel Xeon Phi 7210
All-core turbo (GHz)	2.4	2.4	1.3
Cores	14×2	12×2	64
Host ISA	AVX-2	AVX-512	AVX-512
Memory (GB)	128	96	16 HBM + 96 DDR
	Software		
Operating System	Debian 8, Linux 4.9.0		
Compiler	Intel 19.0.2		

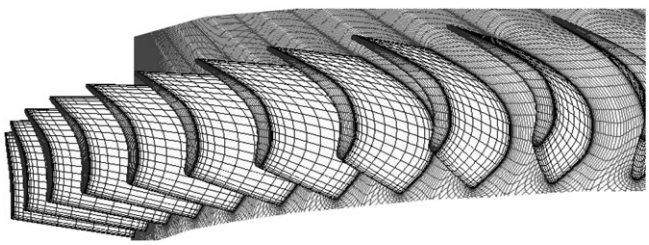

FIGURE 2 Visualization of a rotor section from NASA's SSME two-stage fuel turbine. Blade geometry is similar to the mesh we use

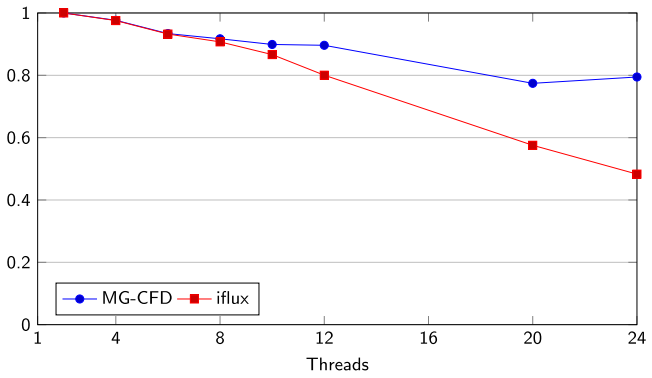

FIGURE 3 Parallel efficiency of MG-CFD and `iflux` on Xeon Skylake with AVX-512 auto-vectorization

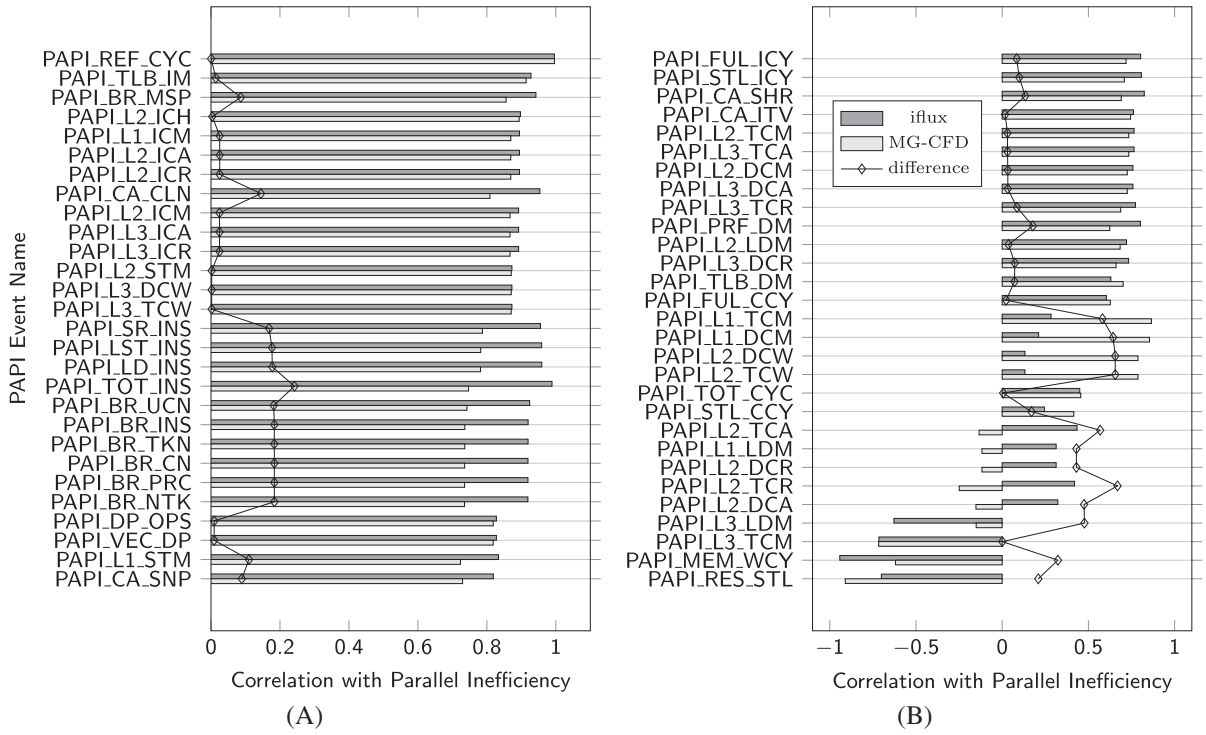

FIGURE 4 Comparison between MG-CFD and `iflux` of their correlation between PAPI preset performance counters and parallel inefficiency

This indicates that the corresponding hardware activity that triggers the counter has a strong influence on scaling performance, where the correlation between a performance counter and parallel efficiency loss is greater than 0.8. The three events PAPI_L1_STM, PAPI_L2_STM, and PAPI_CA_CLN measure Read For Ownership (RFO) events for cache levels 1, 2, and 3, respectively, for which the correlations are strong. In the context of unstructured compute, this is an indication that the memory hierarchy is less able to adequately prefetch the destination arrays in advance of the indirect writes at higher thread counts. This in turn is an indication of contention in the memory hierarchy that is present with both codes. Other notable events are PAPI_SR_INS, PAPI_LST_INS, and PAPI_LD_INS, which count store and load micro-operations and so is another indication of pressure on the memory hierarchy.

6 | PERFORMANCE PREDICTION MODEL

The intended use case for MG-CFD is to assess the impact of new architectures and optimizations to HYDRA without necessarily executing HYDRA on these. As HYDRA is a sensitive code subject to commercial distribution restrictions, MG-CFD can assist in benchmarking, hardware evaluation, and procurement decisions. A focus on individual HYDRA kernels is maintained, as accurate kernel performance predictions can be passed into the HYDRA performance model, which then predicts total walltime based on the predicted execution time of each kernel.[3]

6.1 | Model development

To achieve accurate assessment of hardware and optimizations requires a model of the performance difference between MG-CFD and `iflux`. This model considers performance in terms of clock cycle consumption, termed C_{mini} for MG-CFD and C_{iflux} for `iflux`. To assist in the prediction of scaling performance, the model will focus on predicting the cycle consumption of a single loop iteration of `iflux`, termed $C_{l,iflux}$ for `iflux`, from the empirical measurement of $C_{l,mini}$ of MG-CFD. Assuming that $C_{l,iflux}$ has been estimated, then runtime prediction of a single call to `iflux` at a thread count T is formulated as

$$\text{runtime} = \frac{\max_{t=1}^{T}[C_{l,iflux} \cdot iters_t]}{Hz}. \tag{3}$$

The number of loop iterations performed by each thread, $iters_t$, is considered independent of hardware and is therefore based on prior knowledge. Processor frequency Hz is calculated from MG-CFD's measurements of cycle consumption and runtime.

A simple approach for predicting $C_{l,iflux}$ is to assume that the ratio of $C_{l,iflux}$ to $C_{l,mini}$, defined as R_c, is constant across architectures and ISAs (ignoring bounds imposed by memory performance). Then, $C_{l,iflux}$ is predicted as

$$C_{l,iflux} = R_c C_{l,mini}. \tag{4}$$

It will be shown that the assumption of constant R_c does not always hold true. Thus, a working model must focus on the change in instruction content, and how this causes the observed change in cycle consumption. An additional approach is to assume that cycle consumption is directly proportional to the number of instructions executed, the latter termed I_{iflux} for `iflux` and I_{mini} for MG-CFD. This is equivalent to assuming that `iflux` executes at the same overall instructions-per-cycle (IPC) rate as MG-CFD. This provides the following formulation for $C_{l,iflux}$:

$$C_{l,iflux} = C_{l,mini} \frac{I_{iflux}}{I_{mini}}. \tag{5}$$

6.1.1 | Superscalar extension

Analysis of the compiler-generated assembly files of MG-CFD and `iflux` identifies that they differ in the proportion of particular categories of instructions. We define four categories by throughput and type: low-throughput floating-point (division and square root), high-throughput floating-point, integer, and memory data stores. Different proportions of high and low throughput floating-point operations are likely to result in the two codes having different overall IPC rates. Thus, it is sensible to assume that `iflux` and MG-CFD execute at different IPC rates. Our instruction categorization is applied to MG-CFD to produce $\mathbf{I}_{mini}$, a vector of length 4 where $\mathbf{I}_{i,mini}$ is the number of instructions in the ith category. It is also applied to `iflux` to produce $\mathbf{I}_{iflux}$. Then, the difference in instruction content between the two codes, termed the vector $\Delta\mathbf{I}$, is the element-wise difference of $\mathbf{I}_{mini}$ and $\mathbf{I}_{iflux}$.

To predict the resulting change in cycle consumption of $\Delta\mathbf{I}$ requires a model of superscalar execution to reflect the complexity in modern architectures. A simple model is initially adopted in that each instruction category is scheduled to a single dedicated execution port, and each category is executed in parallel with, and independently of, other categories. The change in instruction content is assumed to be large enough such that when added to a kernel, the compiler is able to optimize the placement of individual instructions to maintain instruction level parallelism (ILP), and so we ignore inter-instruction dependencies. Throughput information is encoded in the vector $\mathbf{c}$ of length 4, where $\mathbf{c}_i$ is the cycles-per-instruction (CPI) estimate for category i (later, we detail how this estimation is calculated). The predicted change in total cycle consumption between MG-CFD and `iflux` is the maximum of cycle consumption of each category, formulated as

$$\Delta C = \max_{i=1}^{4}[(\Delta\mathbf{I}_i)\mathbf{c}_i]. \tag{6}$$

Then, $C_{l,iflux}$ is given by

$$C_{l,iflux} = \frac{C_{mini} - \Delta C}{iters}. \tag{7}$$

6.2 | Contention extension

We extend the model further by considering hardware contention between different instruction categories. Figure 5 shows the portion of the Skylake microarchitecture pipeline related to instruction scheduling. It shows five ports: four of these receive integer or floating-point instructions, and the fifth receives memory data store instructions. Technically, these ports receive micro-ops, but for simplicity, we refer to these as instructions. There are additional ports excluded from the diagram as they are not relevant to the performance difference between MG-CFD and `iflux`. On each clock cycle, the scheduler can assign at most one instruction to each port, then on the next clock cycle, these move onto appropriate execution units. Ports 0 and 1 can receive both integer and floating-point instructions, revoking the prior assumption that

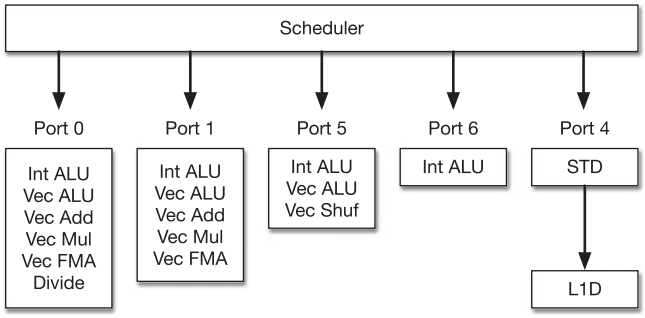

FIGURE 5 Instruction scheduling in Xeon Skylake

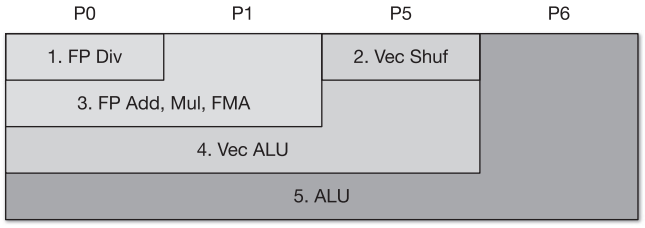

FIGURE 6 Ideal model of Xeon Skylake instruction scheduling

each instruction category is scheduled to a dedicated port. Thus, there is the possibility of contention between different instruction categories. Accordingly, we extend the model to capture this contention while retaining a high degree of flexibility.

To implement this task, modeling makes two assumptions. The first is that, while an execution unit is occupied, then so is its resident port, blocking all other execution units on it. The second assumption is of an ideal instruction scheduler that can schedule in bulk all instructions, scheduling first those instructions with the fewest compatible ports and minimizing the maximum clock cycle consumption across the ports. This process is visualized in Figure 6. The previous "integer" instruction category is separated into three, ALU, Vec ALU, and Vec Shuf, to ensure that member instructions are scheduled to the same ports as well as having similar throughput.

As with the previous model, this model seeks to predict the change in performance that results from the change in instructions from MG-CFD to match `iflux`. The first stage is to predict how those modified instructions were scheduled to ports, using the previously stated assumptions. This produces an allocation matrix **A** with the following structure:

$$\mathbf{A} = \begin{array}{c} \begin{matrix} DIV & VecShuf & STD & FP & VecALU & ALU \end{matrix} \\ \begin{bmatrix} d & – & – & f_2 & a_3 & i_4 \\ – & – & – & f_1 & a_2 & i_3 \\ – & – & – & – & a_1 & i_2 \\ – & v & – & – & – & i_1 \\ – & – & s & – & – & – \end{bmatrix} \end{array} \begin{matrix} P0 \\ P1 \\ P5 \\ P6 \\ P4 \end{matrix}.$$

Our ideal instruction scheduler fills this matrix from left to right, prioritizing those instruction categories with the fewest available ports, seeking to equalize cycle consumption across the ports. Port cycle consumption is calculated by performing a matrix-vector multiply between **A** and the CPI vector **c** (now extended to accommodate the two extra integer instruction categories). This produces the vector **p**, where $\mathbf{p}_i$ is the clock cycle consumption of port i

$$\mathbf{p} = \mathbf{A}\mathbf{c}. \tag{8}$$

Then, the overall change in clock cycle consumption is taken as the maximum port cycle consumption, where N_p is the number of ports

$$\Delta C = \max_{i=1}^{N_p} \mathbf{p}_i. \tag{9}$$

6.3 | CPI estimation

The CPI values are treated as unknowns, as may be the case when evaluating novel architectures. To estimate these, we use eleven distinct variants of MG-CFD's inviscid routine that result from the combinatorial enabling of four arithmetic optimizations missed by the Intel compiler. These variants contain different quantities of division, square-root, all other floating-point operations, integer operations, and register spills while producing the same numerical result. The difference between the most and least expensive variant is equivalent to a difference of 43 AVX2 instructions per loop iteration. To extend this range, we create an additional kernel, derived from the inviscid routine but with ≈ 50% arithmetic instructions removed. This resulting kernel does not correctly implement the inviscid flux accumulation, requiring another correct variant to be executed to maintain solver convergence; however, it enables the range to be extended to 122 instructions. We find that this greater

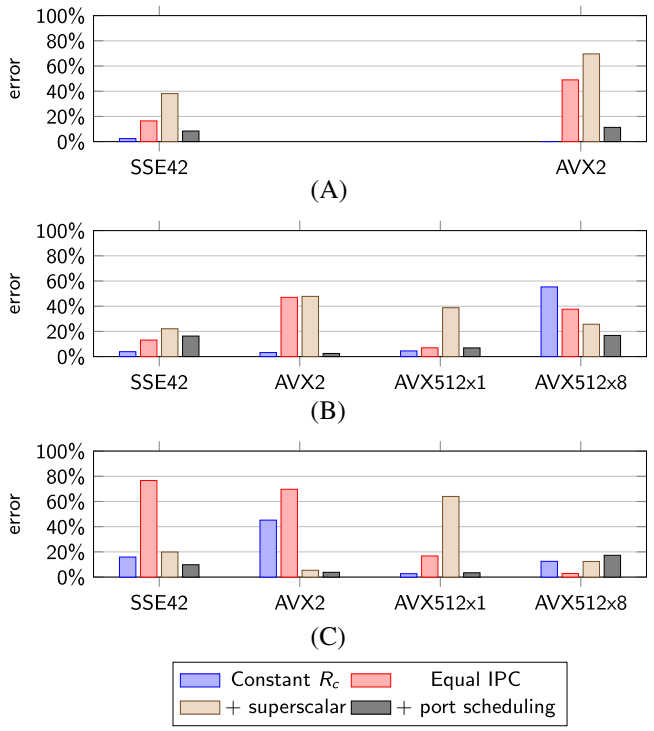

FIGURE 7 Prediction error of described models of iflux cycle consumption. A, Xeon Broadwell; B, Xeon Skylake; C, Xeon KNL

TABLE 3 Single-thread `iflux` runtime prediction errors of the four described projection models

Model	Mean (%)	SD (%)	Worst (%)
Constant R_c	15.6	20.4	55.8
Equal IPC	31.0	25.4	74.7
+ superscalar	34.4	21.3	69.6
+ contention	10.2	6.2	17.8

range produces better CPI estimates that greatly reduce prediction error of $C_{l,iflux}$. To estimate CPI values, we apply basin-hopping optimization, constrained to minimum bounds of 1.0, fitting Equations (9) and (6) to the performance data of these MG-CFD variants.

6.4 | Models comparison

We evaluate each model by its ability to accurately predict single-threaded cycle consumption of `iflux`. We assess across Xeon Skylake, Broadwell and KNL, and across the ISAs AVX512, AVX2, and SSE42. R_c is measured for Broadwell AVX2 and then assumed to be constant for all other combinations of systems and ISAs.

Figure 7 presents accuracies of each model, with accompanying statistics listed in Table 3. The constant R_c model has very high accuracy on most system configurations, averaging 15.6%, but there are two significant exceptions. After enabling AVX-512 vectorization on Skylake, the prediction error increases from 5.2% to 55.8%, caused by the addition of a large quantity of instructions to both kernels that skews R_c toward 1.0. Prediction error for AVX2 on Knights Landing (KNL) is 49.4%, whereas for the same instruction set architecture (ISA) on Skylake and Broadwell, it is near zero. Although this model often generates predictions with error below 16%, occasionally the error is significantly greater, which undermines model reliability. The `equal IPC` model has a high mean error of 31.0% and the highest worst-case error of 74.7%, clear evidence that MG-CFD and `iflux` can and often execute at different overall IPC rates. The superscalar model has a similarly high error, demonstrating that the assumption that each instruction category executes entirely in parallel is incorrect. The final model that incorporates both superscalar execution and instruction scheduling generates predictions with the least mean error of 10.2% with SD 6.2%, and the lowest worst-case error of 17.8%. Overall, our technique of port scheduling delivers reliable results across architecture types.

6.5 | Predicting strong scaling

A necessary component for predicting scaling performance of any kernel in a single node is accounting for limits imposed by the memory hierarchy. This is measured empirically by a new data throughput (DT) kernel, which performs the same data movement as `iflux` but with minimal arithmetic (the least possible without the compiler optimizing away the data accesses). This kernel is executed at the target thread count

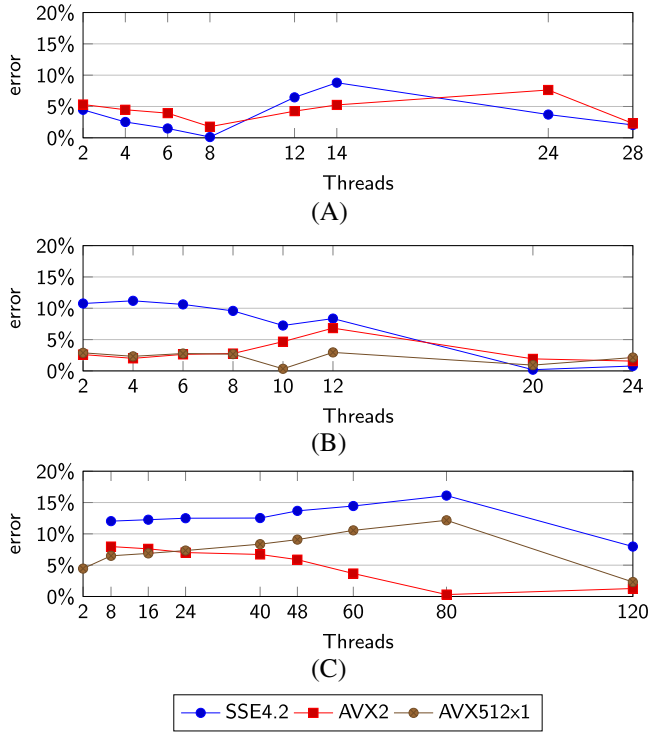

FIGURE 8 Model prediction errors of iflux strong scaling. A, Xeon Broadwell; B, Xeon Skylake; C, Xeon KNL DDR

to produce its cycle consumption $C_{dt,t}$. As this performs minimal computation, it may run at a different turbo clock frequency than MG-CFD, so scaling of $C_{dt,t}$ is necessary to produce a lower bound for $C_{iflux,t}$. This is formulated in Equation (10) to produce the lower bound $C_{min,t}$.

$$C_{min,t} = C_{dt,t} \frac{GHz_{mini,t}}{GHz_{dt,t}}. \tag{10}$$

Figure 8 presents predictions of `iflux` strong scaling on a Skylake node with 1-way SMT, a Broadwell node with 1-way SMT, and a KNL with 2-way SMT, summing runtime across the four MG levels. Prediction error is lowest on the Broadwell node, never exceeding 10% with mean errors of 3.7% for SSE4.2 and 4.4% for AVX2. Prediction error is lower on the Skylake for the AVX-2 and AVX-512 instruction sets, averaging 2.5%, but for SSE4.2, the model error of predicting compute-bound performance increases to ≈ 10%. Prediction error on KNL is low at low thread counts but increases gradually to a peak at 80 threads of 16.1% for SSE4.2 and 12.2% for AVX2. This indicates that, as `iflux` scales up to near the thread count at which it becomes fully bandwidth-bound, it is partially and increasingly limited by memory performance rather than the bound acting in a binary manner as the model assumes. Once `iflux` exceeds 80 threads, it becomes fully memory-bound under AVX2 and AVX-512 with error falling to near-zero, and under SSE4.2 `iflux`, it is very close to this limit as evidenced by the error falling by half to 8%. This discrepancy is explained by newer instruction sets having more sophisticated instructions that perform more operations simultaneously, such as fused multiply-accumulate (FMA) and gather/scatter instructions, that allow a compiler to reduce arithmetic intensity of loops.

6.6 | Predicting performance of HYDRA

Having validated the predictive ability of the MG-CFD and performance model, we now direct attention to the most significant HYDRA kernel, `vflux`. This is the single most expensive loop in HYDRA; for 28 MPI processes on Xeon Broadwell, it accounts for 35.8% of the walltime. Accordingly, its arithmetic intensity is several times that of MG-CFD, posing a significant challenge to our projection model. In contrast, its data access pattern is very similar to that of `iflux`, performing the same single loop over edges, and only differing significantly in the quantity of data associated with each node (cell) (a reflection of the increased complexity of equations needed for viscous and turbulent flow).

For this prediction task, we use a different dataset that is typical for a HYDRA workload, the NASA `Rotor 37` mesh of an axial compressor rotor.[32] This contains ≈ 8.1 M nodes and ≈ 24 M edges, with an additional three MG meshes that result in a total count of ≈15.7 M nodes and ≈ 53 M edges. Thus, this is a significant size for assessing single-node performance.

As performed for the `iflux` predictions, we extract the assembly code of the `vflux` loop from the compiler-generated object file and categorize its constituent instructions. The set of MG-CFD variants are executed on each target system, providing empirical data for estimation of CPI rates. The projection model is applied to provide an estimate of $C_{l,vflux}$, the cycle consumption of a single `vflux` loop iteration. The MG-CFD also measures clock speed, allowing $C_{l,vflux}$ to be converted into *grind time*, the runtime of a single loop iteration. The grind time is passed into

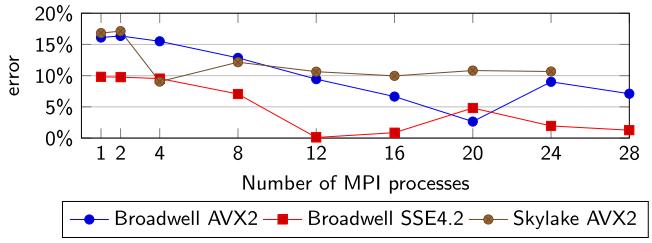

FIGURE 9 Prediction error of HYDRA vflux() compute strong scaling

TABLE 4 Model prediction errors of `vflux` compute strong scaling

System	Mean (%)	SD (%)	Worst (%)
Broadwell AVX2	10.6	4.8	16.3
Broadwell SSE4.2	5.0	4.1	9.8
Skylake AVX2	12.2	3.1	17.1

a pre-existing performance model of HYDRA, which combines it with knowledge of mesh partitioning and a function call trace to produce a prediction of total compute runtime for each HYDRA kernel.[3]

Predictions are made of MPI strong scaling of `vflux` compute on a Xeon Broadwell node and a Xeon Skylake node, using AVX-2 and SSE4.2 instructions sets, presented in Figure 9, with accompanying statistics listed in Table 4. Prediction error for Broadwell and SSE4.2 is the least, achieving mean prediction error of 5.0% and not exceeding 10%. At full node utilization, the error falls to 1.3%. Switching to AVX2 on Broadwell, the prediction error increases to ≈ 16% for low process counts, but reduces steadily as process count increases to a minimum of 2.6% at 20 processes. At full node utilization, the prediction error is 7.1%. On Skylake with AVX2, prediction error is similar as on Broadwell AVX2 for low process counts, and initially shows the same trend of decreasing with increasing process count, but at 12 processes and above the error stabilizes at ≈ 10.5%.

We conclude by focusing on prediction error at the maximum process count on each system. System procurement decision-making typically considers performance in terms of fully-utilized nodes, so achieving accurate predictions of these is particularly important. Our model predicts fully-utilized Broadwell node performance with error 7.1% for AVX2 and 1.3% for SSE4.2, and fully-utilized Skylake node performance with error 10.7%, thus our model has the capability to meaningfully inform procurement decisions with high accuracy.

7 | CONCLUSIONS

This paper reports the development of MG-CFD, the only multigrid unstructured finite-volume CFD mini-application. The MG-CFD has been developed as part of a long-standing university/industry collaboration and, as a result, is the representative of the Rolls-Royce plc production code HYDRA, their primary CFD code used for turbomachinery design.

We applied two mini-application validation techniques, demonstrating that MG-CFD is similar to HYDRA's `iflux` routine. Further analysis highlighted that the scaling behavior achieved was similar, and that the hardware was being stressed in a similar way by both `iflux` and MG-CFD according to hardware counters.

In addition, we construct an analytical performance projection model, targeted toward predicting the performance difference between MG-CFD and HYDRA. This enables projection from MG-CFD to HYDRA performance on a range of existing and emerging HPC architectures. This is highly significant for Rolls-Royce plc as they increase their use of virtual certification and simulation-based engine design.[13] We also demonstrate that it is possible to use a mini-application and performance modeling to predict the performance of a production "target" code, with a mean error of 9.2% for strong-scaling studies.

In future research, we plan to add MPI functionality to MG-CFD through integration of the OP2 library. After validating that multi-node strong scaling performance is similar to HYDRA, we intend to use MG-CFD to explore alternative communication patterns such as partitioning-aware rank placement.

The source code for MG-CFD and scripts for assembly analysis and projection modeling, are available as open-source software on Github.*†‡

ACKNOWLEDGMENTS

This research is supported by Rolls-Royce plc, by the EU Horizon 2020 Clean Sky Project, by the UK Engineering and Physical Sciences Research Council (EPSRC), and by the Intel Corporation: (EP/S005072/1 - Strategic Partnership in Computational Science for Advanced Simulation and

*https://github.com/warwick-hpsc/MG-CFD-app-plain
†https://github.com/warwick-hpsc/MG-CFD-performance-model
‡https://github.com/warwick-hpsc/assembly-loop-extractor

Modelling of Engineering Systems - ASiMoV; EPSRC Industrial CASE award 15220082). The authors would like to thank Rolls-Royce plc for granting permission to publish this work.

ORCID

A. M. B. Owenson https://orcid.org/0000-0003-0397-9448
S. A. Wright https://orcid.org/0000-0001-7133-8533

REFERENCES

1. Mudalige GR, Vernon MK, Jarvis SA. A plug-and-play model for evaluating wavefront computations on parallel architectures. In: Proceedings of the 22nd International Parallel and Distributed Processing Symposium 2008 (IPDPS); 2008; Miami, FL.
2. Barker KJ, Davis K, Hoisie A, et al. Using performance modeling to design large-scale systems. *Computer*. 2009;42(11):42-49.
3. Bunt RA, Pennycook SJ, Jarvis SA, Lapworth L, Ho YK. Model-led optimisation of a geometric multigrid application. Paper presented at: 2013 IEEE 10th International Conference on High Performance Computing and Communications and 2013 IEEE International Conference on Embedded and Ubiquitous Computing; 2013; Zhangjiajie, China.
4. Bunt RA, Wright SA, Jarvis SA, Street M, Ho YK. Predictive evaluation of partitioning algorithms through runtime modelling. Paper presented at: 2016 IEEE 23rd International Conference on High Performance Computing (HiPC); 2016; Hyderabad, India.
5. Hammond SD, Mudalige GR, Smith JA, Jarvis SA, Herdman AJ, Vadgama A. WARPP: a toolkit for simulating high-performance parallel scientific codes. In: Proceedings of the 2nd International Conference on Simulation Tools and Techniques (ICSTT); 2009; Rome, Italy.
6. Janssen CL, Adalsteinsson H, Cranford S, et al. A simulator for large-scale parallel computer architectures. *Int J Distributed Syst Technol*. 2010;1(2):57-73.
7. Pennycook SJ, Hughes CJ, Smelyanskiy M, Jarvis SA. Exploring SIMD for molecular dynamics, using Intel® Xeon® processors and Intel® Xeon Phi coprocessors. In: Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing (IPDPS); 2013; Boston, MA.
8. Tramm JR, Siegel AR, Islam T, Schulz M. XSBench-the development and verification of a performance abstraction for Monte Carlo reactor analysis. In: Proceedings of the Role of Reactor Physics Toward a Sustainable Future (PHYSOR); 2014; Kyoto, Japan.
9. Reguly IZ, Mudalige GR, Giles MB. Design and development of domain specific active libraries with proxy applications. In: Proceedings of the 2015 IEEE International Conference on Cluster Computing (CLUSTER); 2015; Chicago, IL.
10. Mallinson AC, Jarvis SA, Gaudin WP, Herdman AJ. Experiences at scale with PGAS versions of a hydrodynamics application. In: Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models (PGAS); 2014; Eugene, OR.
11. Heroux M, Barrett R. Mantevo project. 2016. https://mantevo.org/. Accessed March 3, 2016.
12. UK mini-app consortium. 2016. http://uk-mac.github.io/papers.html. Accessed March 6, 2016.
13. Strategic partnership in computational science for advanced simulation and modelling of engineering systems - ASiMoV. 2018. https://gtr.ukri.org/projects?ref=EP/S005072/1
14. Owenson A, Wright SA, Jarvis SA, Bunt RA, Ho YK, Street M. Developing and using a geometric multigrid, unstructured grid mini-application to assess many-core architecture. In: Proceedings of the 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP); 2018; Cambridge, UK.
15. ECP proxy apps suite. https://proxyapps.exascaleproject.org/ecp-proxy-apps-suite/. Accessed March 13, 2019.
16. Karlin I, Bhatele A, Keasler J, et al. Exploring traditional and emerging parallel programming models using a proxy application. In: Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing (IPDPS); 2013; Boston, MA.
17. Corrigan A, Camelli FF, Löhner R, Wallin J. Running unstructured grid-based CFD solvers on modern graphics hardware. *Int J Numer Methods Fluids*. 2011;66(2):221-229.
18. Adams MF, Brown J, Shalf J, Van Straalen B, Strohmaier E, Williams S. *HPGMG 1.0: A Benchmark for Ranking High Performance Computing Systems*. Technical Report. Berkeley, CA: Lawrence Berkeley National Laboratory; 2014.
19. Messer OEB, D'Azevedo E, Hill J, Joubert W, Laosooksathit S, Tharrington A. Developing miniapps on modern platforms using multiple programming models. In: Proceedings of the 2015 IEEE International Conference on Cluster Computing (CLUSTER); 2015; Chicago, IL.
20. Sharkawi S, DeSota D, Panda R, et al. Performance projection of HPC applications using SPEC CFP2006 benchmarks. Paper presented at: 2009 IEEE International Symposium on Parallel and Distributed Processing; 2009; Rome, Italy.
21. Hoste K, Phansalkar A, Eeckhout L, Georges A, John LK, De Bosschere K. Performance prediction based on inherent program similarity. In: Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques (PACT); 2006; Seattle, WA.
22. Lapworth L. HYDRA-CFD: a framework for collaborative CFD development. In: Proceedings of the International Conference on Scientific and Engineering Computation (IC-SEC); 2004; Singapore.
23. Spalart PR, Allmaras SR. A one-equation turbulence model for aerodynamic flows. *Recherche Aerospatiale*. 1994;1:5-21.
24. Moinier P, Müller J, Giles MB. Edge-based multigrid and preconditioning for hybrid grids. *AIAA Journal*. 2002;40(10):1954-1960.
25. Martinelli L, Jameson A. Validation of a multigrid method for the Reynolds averaged equations. Paper presented at: AIM 26th Aerospace Sciences Meeting; 1988; Reno, NV.
26. Trottenberg U, Oosterlee CW, Schuller A. *Multigrid*. Amsterdam, The Netherlands: Elsevier; 2001.
27. Che S, Boyer M, Meng J, et al. Rodinia: a benchmark suite for heterogeneous computing. Paper presented at: 2009 IEEE International Symposium on Workload Characterization (IISWC); 2009; Austin, TX.
28. Briggs WL. *A Multigrid Tutorial*. Philadelphia, PA: Society for Industrial and Applied Mathematics; 1987.
29. Hodson HP, Dominy RG. Three-dimensional flow in a low-pressure turbine cascade at its design condition. *J Turbomach*. 1987;109(2):177-185.
30. NASA. TCGRID v. 400. NASA. https://www.grc.nasa.gov/www/5810/rvc/tcgrid.htm. Accessed November 8, 2017.

31. Browne S, Deane C, Ho G, Mucci P. Papi: a portable interface to hardware performance counters. In: Proceedings of the Department of Defense HPCMP Users Group Conference; 1999.
32. Reid L, Moore RD. *Design and Overall Performance of Four Highly Loaded, High Speed Inlet Stages for an Advanced High-Pressure-Ratio Core Compressor*. Technical Report. Cleveland, OH: NASA; 1978.

How to cite this article: Owenson AMB, Wright SA, Bunt RA, Ho YK, Street MJ, Jarvis SA. An unstructured CFD mini-application for the performance prediction of a production CFD code. *Concurrency Computat Pract Exper*. 2019;e5443. https://doi.org/10.1002/cpe.5443